МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Задача Коммивояжёра
МВиГ: Алгоритм Литтла с модификацией
Приближённый алгоритм: АБС
Вариант – 2

Студент гр. 3388 _____ Березовский М.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы

Изучить задачу Коммивояжёра, МВиГ, Алгорит Литтла, приближённые алгоритмы решения. Применить полученные знания на практике, реализовав программный код.

Задание

Решить ЗК двумя методами в соответствии с вариантом: 1) Методом ВиГ. 2)Приближённым методом.
Дано: матрица весов графа, все веса неотрицательны; стартовая вершина.
Найти: путь коммивояжёра (последовательность вершин) и его стоимость.

2 МВиГ: Алгоритм Литтла с модификацией: после приведения матрицы, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. Приближённый алгоритм: АБС.
Замечание к варианту 2: Начинать АБС со стартовой вершины.

Выполнение работы

Описание алгоритма для решения задачи

Метод ветвей и границ: Алгоритм Литтла с модификацией.

Задача коммивояжёра (Traveling Salesman Problem, TSP) — это классическая NP-трудная задача комбинаторной оптимизации, в которой необходимо найти кратчайший маршрут, проходящий через все вершины графа ровно один раз и возвращающийся в начальную точку.
Реализованный алгоритм использует метод ветвей и границ (Branch and Bound), а конкретнее алгоритм Литтла с модификацией, который позволяет отсекать неоптимальные маршруты и минимизировать перебор возможных вариантов.

Основные этапы алгоритма

Редукция матрицы расстояний (reduce_matrix) - Из каждой строки и столбца вычитается минимальный элемент, чтобы в каждой строке и столбце присутствовал хотя бы один ноль.
Это приводит к снижению общей стоимости решения и формирует нижнюю границу стоимости пути.

Выбор ребра с наибольшей оценкой (get_greatest_zero) - Среди нулей в редуцированной матрице выбирается наилучшее ребро, основываясь на сумме минимальных элементов в соответствующей строке и столбце. Этот выбор минимизирует будущие увеличения стоимости маршрута.

Расчёт нижней границы с использованием минимального остовного дерева (prim_mst, calculate_bound) - Оценка стоимости оставшегося маршрута производится с помощью минимального остовного дерева (МОД), что даёт более точную нижнюю границу.

Ветвление и поиск (branching) - Рекурсивно строится дерево решений, в котором:

Одно направление ветвления принимает выбранное ребро,

Другое — запрещает его, проверяя, есть ли более дешёвый путь.

Если в процессе поиска текущая стоимость пути превышает текущий best_cost, ветка отсекается.

Перебор всех возможных стартовых вершин (solve_little) - Запуск алгоритма для каждого возможного начального города и поиск глобального оптимального решения.

АБС:

Алгоритм ближайшего соседа (АБС) - это жадный алгоритм, который строит приближённое решение задачи коммивояжёра. В отличие от алгоритма Литтла, он не гарантирует нахождение оптимального маршрута, но выполняется за полиномиальное время и подходит для больших входных данных.

Основные этапы алгоритма

Инициализация - Задаётся начальный город start. Создаётся множество непосещённых городов unvisited_vertexes, содержащее все вершины, кроме стартовой. Начальный маршрут route содержит только стартовую вершину. Общая стоимость пути path_cost инициализируется нулём.

Жадный выбор ближайшего города - Пока остаются непосещённые вершины, выбирается город с минимальным расстоянием от текущего (cur). Этот город добавляется в маршрут, а его расстояние к path_cost. Удаляется из множества unvisited_vertexes.

Возврат в стартовую вершину - Когда все города пройдены, маршрут замыкается, добавляя start в конец. Финальная стоимость маршрута увеличивается на расстояние от последнего посещённого города до стартового. Вывод результата - Возвращается найденный маршрут и его стоимость.

Оптимизация алгоритма.

Метод ветвей и границ позволяет значительно сократить количество перебираемых вариантов, но требует эффективного управления структурой данных и оценкой нижней границы. В коде используется несколько оптимизаций:

Редукция матрицы уменьшает вычислительные затраты, сразу снижая стоимость пути.

Использование минимального остовного дерева (МОД) в оценке нижней границы помогает быстрее отсекать заведомо невыгодные маршруты.

Глубина рекурсии ограничивается отсечением по стоимости, что предотвращает избыточное ветвление.

Работа с матрицами через NumPy, что значительно ускоряет обработку.

Оценка сложности алгоритма во времени и памяти в нотации О.

МВиГ:

Редукция матрицы выполняется за $O(n^2)$, так как требуется пройтись по n строкам и n столбцам.

Поиск наилучшего нуля выполняется за $O(n^2)$, так как нужно проверить все возможные ребра.

Оценка нижней границы с помощью MST в худшем случае выполняется за $O(n^2)$, используем алгоритм Прима.

Рекурсивное ветвление имеет наихудшую сложность O(n!), так как в теории рассматриваются все перестановки городов.

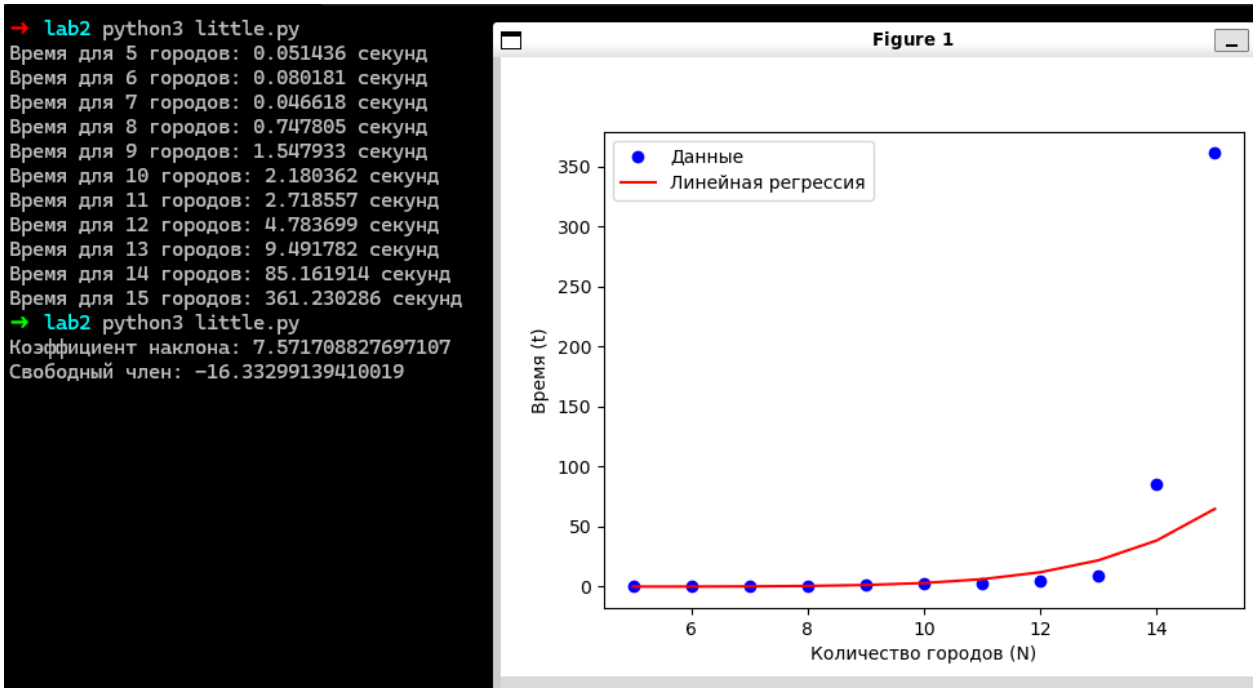
Итого, асимптотическая сложность алгоритма в худшем случае: O(n!)

Оценка по памяти:

Матрица n x n требует $O(n^2)$.

Рекурсивный стек может достигать глубины O(n).

В худшем случае требуется хранить O(n!) состояний, но за счёт отсечения ветвлений это число сильно уменьшается и становится O(k^n)



АБС:

Жадный выбор ближайшего соседа: $O(n^2)$, так как на каждом шаге происходит $O(n)$ перебор оставшихся городов, а шагов $O(n)$.

Общая временная сложность: $O(n^2)$

Оценка по памяти:

Матрица $n \times n$: $O(n^2)$.

Список маршрута $O(n)$.

Множество непосещённых вершин $O(n)$.

Итого, общая сложность по памяти: $O(n^2)$

Тестирование. Демонстрация граничных случаев алгоритма.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии

1.	3 -1 1 3 3 -1 1 1 2 -1	0 1 2 3.0	Верный вывод
2.	4 -1 3 4 1 1 -1 3 4 9 2 -1 4 8 9 2 -1	0 3 2 1 6.0	Верный вывод
3.	6 -1 25 40 31 27 5 -1 17 30 25 19 15 -1 6 1 9 50 24 -1 6 22 8 7 10 -1	0 1 2 4 3 62.0	Верный вывод

Выводы

В зависимости от требований к точности и времени работы, следует выбирать соответствующий алгоритм. Если важно получить оптимальное решение, но допустимо долгое вычисление, то метод Литтла подходит лучше. Если же требуется быстрое приближённое решение на больших данных, то АБС является более практичным вариантом.

Разработанный программный код см. в приложении А

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: little.py

```
import numpy as np

INF = np.inf
best_cost = INF
best_path = []

def reduce_matrix(matrix):
    delta = 0
    reduced_matrix = np.copy(matrix)
    size = reduced_matrix.shape[0]

    for i in range(size):
        row_min = np.min(reduced_matrix[i])
        if row_min == INF:
            continue
        delta += row_min
        reduced_matrix[i] -= row_min

    for j in range(size):
        col_min = np.min(reduced_matrix[:, j])
        if col_min == INF:
            continue
        delta += col_min
        reduced_matrix[:, j] -= col_min

    return reduced_matrix, delta


def get_greatest_zero(matrix, current_node):
    size = matrix.shape[0]
    degrees = []

    for j in range(size):
        if matrix[current_node][j] == 0:
            row_min = np.min(matrix[current_node][np.arange(size) != j])
            row_min = row_min if row_min != INF else 0

            col_min = np.min(matrix[np.arange(size) != current_node, j])
            col_min = col_min if col_min != INF else 0

            degrees.append((current_node, j, row_min + col_min))

    return sorted(degrees, key=lambda x: -x[2])

def prim_mst(submatrix):
    size = submatrix.shape[0]
    if size <= 1:
        return 0

    selected = {0}
```

```
    mst_cost = 0

    while len(selected) < size:
        min_edge = INF
        best_vertex = None

        for u in selected:
            for v in range(size):
                if v not in selected and matrix[u, v] < min_edge:
                    min_edge = matrix[u, v]
                    best_vertex = v
        selected.add(best_vertex)
        mst_cost += min_edge

    return mst_cost

def calculate_bound(pieces, matrix):
    if not pieces:
        return 0

    size = len(pieces)
    submatrix = np.full((size, size), INF)

    for i in range(size):
        for j in range(i + 1, size):
            start_i, end_i = pieces[i]
            start_j, end_j = pieces[j]
            w = min(matrix[end_i][start_j], matrix[end_j][start_i])
            submatrix[i][j] = submatrix[j][i] = w

    return prim_mst(submatrix)

def branching(matrix, path, cost, pieces, start):
    global best_cost, best_path
    n = matrix.shape[0]

    if len(path) == n:
        final_cost = cost + matrix[path[-1]][start]
        if final_cost < best_cost:
            best_cost = final_cost
            best_path = path + [start]
        return

    bound = calculate_bound(pieces, matrix)
    if cost + bound >= best_cost:
        return

    current_node = path[-1]
    zero_degrees = get_greatest_zero(matrix, current_node)

    if not zero_degrees:
        return

    for current, next_node, _ in zero_degrees:
        if next_node in path or matrix[current][next_node] == INF:
            continue
```

```
        new_pieces = pieces.copy()
        merged = False
        for i, (s, e) in enumerate(new_pieces):
            if e == current:
                new_pieces[i] = (s, next_node)
                merged = True
            elif s == next_node:
                new_pieces[i] = (current, e)
                merged = True
        if not merged:
            new_pieces.append((current, next_node))

        new_mat = np.copy(matrix)
        new_mat[current, :] = INF
        new_mat[:, next_node] = INF
        new_mat[next_node, current] = INF

        reduced_mat, reduction_cost = reduce_matrix(new_mat)
        new_cost = cost + matrix[current][next_node] + reduction_cost
        branching(reduced_mat, path + [next_node], new_cost, new_pieces,
start)

        new_mat_right = np.copy(matrix)
        new_mat_right[current][next_node] = INF
        reduced_right, reduction_right = reduce_matrix(new_mat_right)
        branching(reduced_right, path, cost + reduction_right, pieces,
start)
        break


def solve_little(matrix):
    global best_cost, best_path
    all_best_cost = INF
    all_best_path = []
    n = matrix.shape[0]

    for start in range(n):
        best_cost = INF
        best_path = []

        reduced_mat, init_cost = reduce_matrix(matrix)
        branching(reduced_mat, [start], init_cost, [], start)

        if best_cost < all_best_cost:
            all_best_cost = best_cost
            all_best_path = best_path

    return all_best_path, all_best_cost


if __name__ == "__main__":
    n = int(input())
    matrix = []
    for _ in range(n):
        row = list(map(int, input().split()))
```

```
        matrix.append([float('inf') if x == -1 else float(x) for x in
row])


    matrix = np.array(matrix, dtype=np.float64)
    path, cost = solve_little(matrix)
    print(" ".join(map(str, path[:-1])))
    print(cost)
```

Название файла: abs.py

```
import numpy as np

from matrix import *

def tsp_abs(matrix, start):
    N = len(matrix)
    route = [start]
    path_cost = 0
    cur = start
    unvisited_vertexes = set(range(N)) - {start}

    while unvisited_vertexes:
        d = []
        for v in unvisited_vertexes:
            d.append(matrix[cur][v])
              next_vertex = min(unvisited_vertexes, key=lambda v:
matrix[cur][v])

        path_cost += matrix[cur][next_vertex]
        route.append(next_vertex)
        unvisited_vertexes.remove(next_vertex)
        cur = next_vertex

    path_cost += matrix[cur][start]

    route.append(start)

    return route, path_cost


n = int(input("Введите количество городов (N): "))
sym = input("Симметричная матрица? (y/n): ").strip().lower() == 'y'

matrix = generate_matrix(n, symmetric=sym)

save_matrix(matrix, "data.txt")

loaded_matrix = load_matrix("data.txt")
print(loaded_matrix)

start = int(input("Введите индекс стартовой вершины (от 0 до N-1): "))
route, cost = tsp_abs(loaded_matrix, start)

print("Найденный маршрут: ", route)
```

```
print("Стоимость маршрута: ", cost)
```

Название файла: matrix.py

```
import numpy as np

def generate_matrix(n, symmetric=False):
    if symmetric:
        A = np.random.randint(1, 100, size=(n, n)).astype(float)
        A = (A + A.T) // 2
    else:
        A = np.random.randint(1, 100, size=(n, n)).astype(float)
    np.fill_diagonal(A, np.inf)
    return A

def save_matrix(matrix, filename="data.txt"):
    np.savetxt(filename, matrix, delimiter=',', fmt='%g')

def load_matrix(filename="data.txt"):
    return np.loadtxt(filename, delimiter=',')
```